Java SmartMerger

By: Kamden Chew & Robert Kolmos

Motivation:

Merge conflicts in git can often result in trivial but annoying cases. For example, white lines and imports often cause conflicts that the computer is incapable of resolving (as of now) but the human requires no thought to fix. A smarter merge tool could reduce the amount of busy work required freeing up the human to work on the nonobvious parts of the code. This merge tool could suggest changes to the user allowing the user to simply skim the auto generated output instead of having to review each individual change. This would both save time and reduce the hassle associated with conflicts. For the sake of completing this project in a timely manner, we are going to limit the scope of our project to Java files. Although we are only implementing this merging tool for Java, it can also serve as a proof of concept for other programming languages.

Approach:

We have identified a few cases that we believe the computer should be able to resolve accurately. These cases include:

- Differences in white space.
- Differences in import statements.
- Method/variable renamings.
- More complicated but potentially possible are cases where accepting one change disambiguates other changes (i.e. accepting a change to a method header results in only one set of changes to the calls for the same method compiling).

Limitations:

Our approach is limited by the requirement that the output must be sound in cases where the tool does something automatically (i.e. that the automated merging should never resolve a conflict when it is not 100% sure which side should be selected). We make this a requirement since the possibility of the automated system introducing a change into the code that the human does not want is unacceptable. In other cases the tool can suggest a change, but allow the user to override it. This allows the tool to eliminate as much work as possible for the human, while still providing the ability to resolve conflicts as the user sees fit.

Challenges and Risks:

As part of our tool, we would like to add functionality to handle refactorization merge conflicts. Attempts to refactor files could be complex, and if the automatically suggested refactoring were to be unsound, then our tool could make merge conflicts even more annoying than they already are. To minimize the risk of having an unsound merging resolution we will dedicate extra time towards testing our implementation and consulting others to verify that our tool merges files in a way that is functionally equivalent to a manual merge of the same two files.

Examples:

A:	**B:**	**Tool Suggests:**

void foo() {
    System.out.println(“foo”);
}
void bar() {
    System.out.println(“bar”);
}
	void foo() {
	    System.out.println(“foo”);
	}
	
	void bar() {
	    System.out.println(“bar”);
	}
		void foo() {
		    System.out.println(“foo”);
		}
		
		void bar() {
		    System.out.println(“bar”);
		}

A:	**B:**	

void foo() {
    System.out.println(“foo”);
}
 	
void bar() {
    foo();
    System.out.println(“bar”);
}
	void baz() {
	    System.out.println(“foo”);
	}
	
	void bar() {
	    baz();
	    System.out.println(“bar”);
	}
		User accepts foo->baz header change.
		
		Solution:
		void baz() {
		    System.out.println(“foo”);
		}
		
		void bar() {
		    baz();
		    System.out.println(“bar”);
		}

A:	**B:**	

import java.util.ArrayList;
void foo() {
    new ArrayList<String>();
    System.out.println(“foo”);
}
 	
void bar() {
    System.out.println(“bar”);
}
	void foo() {
	    System.out.println(“foo”);
	}
	
	void bar() {
	    System.out.println(“bar”);
	}
		User accepts deletion of the first line of method foo().
		
		Solution:
		void foo() {
		    System.out.println(“foo”);
		}
		
		void bar() {
		    System.out.println(“bar”);
		}